

Nº Orden	Apellido y nombre	L.U.	Cantidad de hojas
			5

## Organización del Computador 2

### Recuperatorio del Primer Parcial — ??/??/17

1 (40)	2 (40)	3 (20)	4 (20)
40	40	10	90

#### Normas generales

- Numere las hojas entregadas. Complete en la primera hoja la cantidad total de hojas entregadas.
- Entregue esta hoja junto al examen, la misma **no** se incluye en la cantidad total de hojas entregadas.
- Está permitido tener los manuales y los apuntes con las listas de instrucciones en el examen. Está prohibido compartir manuales o apuntes entre alumnos durante el examen.
- Cada ejercicio debe realizarse en hojas separadas y numeradas. Debe identificarse cada hoja con nombre, apellido y LU.
- La devolución de los exámenes corregidos es personal. Los pedidos de revisión se realizarán por escrito, antes de retirar el examen corregido del aula.
- Los parciales tienen tres notas: I (Insuficiente): 0 a 59 pts, A- (Aprobado condicional): 60 a 64 pts y A (Aprobado): 65 a 100 pts. No se puede aprobar con A- ambos parciales. Los recuperatorios tienen dos notas: I: 0 a 64 pts y A: 65 a 100 pts.

### Ej. 1. (40 puntos)

En los partidos de basket suelen contabilizarse estadísticas del juego. En particular, vamos a tener una lista con todos los lanzamientos al aro que se realizaron en el juego, los puntos que produjo el lanzamiento (entre 0 y 3 puntos) y en el orden en el que sucedieron.

La estructura puntos indica los puntos producidos por cada lanzamiento, una breve descripción indicando el tipo (bandeja, suspendido, vuelco, libre, etc), un booleando indicando si el lanzamiento fue realizado por el equipo local o el visitante y el número del jugador que realizó dicho intento.

```
typedef struct {
    uint8_t puntos;
    char* descripcion;
    bool local;
    lanzamiento* siguiente;
    uint8_t jugador;
} lanzamiento;
```

Se pide ordenar la lista de lanzamientos al aro de manera que queden primeros los lanzamientos del equipo local y segundo los del equipo visitante. El orden relativo de los lanzamientos del equipo visitante y del local deben mantenerse.

- (6p) Indicar los desplazamientos dentro de la estructura y su tamaño, notar que no es `__packed__`.
- (4p) Plantear la aridad de la función a realizar. Justificar el porqué de los parámetros.
- (12p) Escribir en C la función pedida.
- (18p) Escribir en ASM la misma función pero además eliminando los lanzamientos que no produjeron puntos.

### Ej. 2. (40 puntos)

```
typedef struct {
    uint2_t lanzamiento;
    uint1_t acierto;
    uint1_t equipo;
    uint12_t jugador;
} lanzamiento __attribute__((packed));
```

Donde se usan dos bits para indicar el tipo de lanzamiento (nulo: 00, libre:01, doble:10 o triple:11), un bit para indicar si el lanzamiento entro en el aro, un bit para indicar si el intento fue realizado por el equipo local o el visitante y, por ultimo, 12 bits para indicar el numero del jugador que realizo el lanzamiento.

El largo de los arreglos que procesaremos tendrán largo múltiplo de 8.

- a- (20p) Escribir en ASM una función <sup>uint32\_t</sup> `void cantidad_de_puntos(lanzamiento* lanzamientos, int tamano, uint8_t el_equipo)` que devuelva la cantidad de puntos anotados por el equipo.
- b- (20p) Escribir en ASM una función `float promedio_de_tres(lanzamiento* lanzamientos, int tamano)` que calcule el promedio de los lanzamientos de tres del equipo local.

### Ej. 3. (20 puntos)

Supongamos que tenemos una funcion recursiva que no utiliza registros de la convencion ni variables locales. Cada tanto, esta funcion recursiva llega a un caso base en el cual, por alguna razon, quiere retroceder no un llamado, sino n llamados.

Se pide implementar la funcion `returnene` que en vez de volver al llamado anterior, vuelva n llamados anteriores. Suponer que esos llamados estan en la pila.

- (a) (5p) Dibujar el estado de la pila, justo <sup>antes</sup> despues de llamar a `returnen`, con tres llamados recursivos.
- (b) (15p) Escriba el código ASM de la función `returnen`.  
La aridad de la función será: `void returnene(unsigned int n)`



1) a) type def struct {	Tamaño de los atributos	Offset de los atributos
uint8_t puntos;	→ 1B	→ 0
char* descripcion;	→ 8B	→ 8
bool local;	→ 1B	→ 16
lanzamiento* siguiente;	→ 8B	→ 24
uint8_t jugador;	→ 1B	→ 32
} lanzamiento;		

Notamos que bool ocupa 1B, los punteros ocupan 8B (tamaño de una dirección de memoria en 64b) y que la estructura no es packed, por lo que cada atributo está alineado en su tamaño usando padding si es necesario. El tamaño de la estructura es múltiplo del tamaño de su atributo más grande, quedando:

$$\text{sizeof(lanzamiento)} = 32B + 8B = 40B$$

b) void primLocal Desp Visita (lanzamiento\*\* lista);

La función no asegura de modificar la lista pasada por referencia a través de un doble puntero. En razón por la que pasar el doble puntero a la función es por el hecho de que necesitamos modificar la estructura de la lista, no su contenido. Si la referencia fuera a través de un puntero a lanzamiento (cabe de la lista), luego al recorrerla no sabríamos si el puntero pasado es NULL, no podríamos modificar el lugar del primer elemento, y no podríamos borrarla (lo cual será útil en el punto d)). Esto es porque en el caso por referencia el puntero que sirve como referencia es pasado por copia y modificando no alteramos su contenido fuera de la función.

c) void primLocal Desp Visita (lanzamiento\*\* lista)

lanzamiento** visita;	→ Declara una variable para la lista de visitantes.
lanzamiento* actual = *lista;	→ Declara un iterador de la lista en la primera posición.
lanzamiento* ultLocal = NULL;	→ Declara un puntero al último local en NULL.
lanzamiento* ultVisita = NULL;	→ Declara un puntero al último visitante en NULL.
lanzamiento* refActual = lista;	→ Declara una referencia al iterador de la lista.
while (actual != NULL)	→ Recorremos hasta que actual sea el lanzamiento siguiente al último.
{ if (actual == local)	// Caso lanzamiento del equipo local.
{ if (ultLocal == NULL)	→ Vemos si no hubo otro lanzamiento local recorrido.
{ *lista = *actual;	→ Si no, hago que lista apunte al primer local (actual).
ultLocal = actual;	→ Actualizo el último local con el actual.
refActual = &(actual->siguiente);	} Actualizo actual y su referencia al siguiente.
actual = actual->siguiente;	
} else {	
ultLocal->siguiente = actual;	→ Si hubo un local, hago que su siguiente sea el actual.
ultLocal = actual;	→ Tomo a actual como el último local.
refActual = &(actual->siguiente);	} Avanzo actual y su referencia al siguiente.
actual = actual->siguiente;	
} else {	// Caso visitante.
if (ultVisita == NULL)	→ Verifico análogamente si hubo un visitante anterior (análogo al caso local).
visita = refActual;	→ Si no hubo, actualiza visita y ultVisita, y avanza actual con su referencia al siguiente.
ultVisita = actual;	
refActual = &(actual->siguiente);	
actual = actual->siguiente;	
} else {	
ultVisita->siguiente = actual;	→ Si hubo, tomo al actual como el siguiente del último y actualizo ultVisita,
ultVisita = actual;	avanzando actual y su referencia.
refActual = &(actual->siguiente);	



```
actual = actual->siguiente;
```

```
}
```

```
}
```

```
}
```

```
if (ultVisita != NULL)
```

```
ultVisita->siguiente = NULL;
```

```
if (ultLocal != NULL)
```

```
ultLocal->siguiente = *visita;
```

```
{
```

```
}
```

```
}
```

→ Si hubo algún lanzamiento visitante, hacemos que su siguiente sea NULL (no haya)

→ Si además hubo lanzamiento local, concatenamos el siguiente al último local con el primer visitante (si hay solo locales o solo visitantes no hay concatenación)

d) void primLocal Desp Visita Con Puntos (lanzamiento\*\* lista)

```
lanzamiento** visita;
```

```
lanzamiento* actual = *lista;
```

```
...
```

```
lanzamiento* ultLocal = NULL;
```

```
lanzamiento* ultVisita = NULL;
```

```
lanzamiento** refActual = lista;
```

```
while (actual != NULL)
```

```
{ if (actual->puntos == 0)
```

```
lanzamiento* borrar = actual;
```

```
refActual = &(actual->siguiente);
```

```
actual = actual->siguiente;
```

```
free(borrar->descripcion);
```

```
free(borrar);
```

```
} else {
```

```
    // ← caso normal del ciclo en la función anterior
```

```
}
```

```
}
```

```
if (ultVisita != NULL)
```

```
ultVisita->siguiente = NULL;
```

```
if (ultLocal != NULL)
```

```
ultLocal->siguiente = *visita;
```

```
} else {
```

```
    *lista = *visita;
```

```
}
```

```
} else {
```

```
if (ultLocal != NULL)
```

```
ultLocal->siguiente = NULL;
```

```
} else {
```

```
    *lista = NULL;
```

```
}
```

```
}
```

```
}
```

→ Verifico si el actual tiene puntos

→ Si no, tomo una variable temporal para borrar el lanzamiento

→ Avanzo actual y su referencia al siguiente lanzamiento

→ Libero la descripción del lanzamiento

→ Libero el lanzamiento en sí

← Puede que la lista original empiece en un local sin puntos, por lo que direccionamos lista al contenido de visita

← Puede que tenga todos lanzamientos visitantes sin puntos, por lo que me aseguro que el último local tenga como siguiente a NULL (ninguno)

← caso: todos los lanzamientos de la lista no tienen puntos, entonces la lista quedaría

```
% define OFFSET_LANZ_PTS 0
```

```
% define OFFSET_LANZ_DESC 8
```

```
% define OFFSET_LANZ_LOCAL 16
```

```
% define OFFSET_LANZ_SIG 24
```

```
% define OFFSET_LANZ_SIG 32
```

```
% define NULL 0
```

```
extern free
```

prim Local Desp Visita Con Puntos:

```
push rbp
mov rbp, rbp
```

→ Stack frame

```
push rbx
push r12
push r13
push r14
push r15
```

Preservo registros de la convención en la pila (pila desalineada tras la operación)

```
xor rbx, rbx
xor r12, r12
mov r12, [rdi]
mov r13, NULL
mov r14, NULL
mov r15, rdi
mov rsi, rdi
```

← visita

← actual = \*lista

← ult local

← ult Visita

← refActual

← lista

ciclo:

```
cmp r12, NULL
je .fin_ciclo
jmp .puntos
```

← actual == NULL?

puntos:

```
mov r8, [r12 + OFFSET_LANZ_PTS]
cmp r8, 0
je .sin_puntos
jmp .con_puntos
```

← r8 = actual → puntos

← actual → puntos == 0?

sin\_puntos:

```
push rsi
mov rdi, [r12 + OFFSET_LANZ_DESC]
lea r15, [r12 + OFFSET_LANZ_SIG]
call free
mov rdi, r12
mov r12, [r15]
call free
pop rsi
jmp ciclo
```

← Conservo lista en la pila

← rdi = actual → desc (borrar → desc)

← r15 = &(actual → siguiente)

← Free (borrar → desc)

← r12 contiene actual y lo paso a rdi para borrarlo

← r12 = actual → siguiente

← free (borrar)

← Restaura lista en rsi

con\_puntos:

```
mov r8, [r12 + OFFSET_LANZ_LOCAL]
cmp r8, 1
je .local
jmp .visita
```

← actual → local == 1 == true?

local:

```
cmp r13, NULL
je .primer_local
jmp .nuevo_local
```

← ult local == NULL?

primer\_local:

```
mov [rsi], r12
mov r13, r12
```

← \*lista = actual

← ult local = actual



```

    lea r15, [r12+OFFSET_LANZ_SIG]
    mov r12, [r12+OFFSET_LANZ_SIG]
    jmp .ciclo

```

- nuevo\_local:

```

    mov [r13+OFFSET_LANZ_SIG], r12    → ultLocal → siguiente = actual
    mov r13, r12                      → ultLocal = actual
    lea r15, [r12+OFFSET_LANZ_SIG]
    mov r12, [r12+OFFSET_LANZ_SIG]
    jmp .ciclo

```

- visita:

```

    cmp r14, NULL                      → ultVisita == NULL?
    je .primer-visita
    jmp .nuevo-visita

```

- primer-visita:

```

    mov r15, r15                      → visita = refActual
    mov r14, r12                      → ultVisita = actual
    lea r15, [r12+OFFSET_LANZ_SIG]
    mov r12, [r12+OFFSET_LANZ_SIG]
    jmp .ciclo

```

- nuevo-visita:

```

    mov [r14+OFFSET_LANZ_SIG], r12    → ultVisita → siguiente = actual
    mov r14, r12                      → ultVisita = actual
    lea r15, [r12+OFFSET_LANZ_SIG]
    mov r12, [r12+OFFSET_LANZ_SIG]
    jmp .ciclo

```

- fin\_ciclo:

```

    cmp r14, NULL                      → ultVisita == NULL? (hubo visitantes con puntos?)
    je .no-visita
    jmp .hay-visita

```

- hay\_visita:

```

    mov qword [r14+OFFSET_LANZ_SIG], NULL → ultVisita → siguiente = NULL
    cmp r13, NULL                      → ultLocal == NULL? (hubo locales con puntos?)
    je .solo-visitas
    jmp .local-visitas

```

- local\_visitas:

```

    mov r8, [r13]
    mov [r13+OFFSET_LANZ_SIG], r8      → ultLocal → siguiente = *visita
    jmp .end

```

- solo\_visitas:

```

    mov r8, [r13]
    mov [rsi], r8                      → *lista = *visita
    jmp .end

```

- no\_visita:

```

    cmp r13, NULL                      → ultLocal == NULL? (Como no hay visitantes con puntos, ¿hubo locales con puntos?)
    je .nada
    jmp .solo_locales

```

- solo\_locales:

```

    mov qword [r13+OFFSET_LANZ_SIG], NULL → ultLocal → siguiente = NULL
    jmp .end

```

.nada:

mov qword [rsi], NULL → \*lista = NULL (lista vacía)

.end:

pop r15

pop r14

pop r13

pop r12

pop rbx

pop rbp

ret

} Recupero los valores de los registros, terminando el stack frame



2 a) Como que equipo es el local, lanz el lanzamiento, acierta o no, equipos equipos y jug el jugador (atributos de lanzamiento)

cantidad de puntos:  $\rightarrow \text{uint32_t cantidad\_de\_puntos}(\text{lanzamientos} * \text{lanzamientos}, \text{int tamaño}, \text{uint8_t el\_equipo})$   
 push rbp  
 mov rbp, rsp  
 xor rax, rax  $\rightarrow$  Suma de puntos anotados por el equipo  
 pxor xmm0, xmm0  
 pxor xmm1, xmm1  
 mov ecx, esi  $\rightarrow$  ecx = iterador del ciclo  
 shr ecx, 3  $\rightarrow$  tamaño múltiplo de 8  
 qsrst xmm1, edx, 0  $\rightarrow$  xmm1 = [0, 0, 0, ..., dl]  
 psrshw xmm1, xmm1, 0  $\rightarrow$  xmm1 = [0, 0, ..., 0 dl, dl, ..., dl]  
 movdqu xmm2, xmm1  
 psllq xmm2, 8  $\rightarrow$  xmm2 = [dl, dl, ..., dl, 0, 0, ..., 0]  
 paddw xmm1, xmm2  $\rightarrow$  xmm1 = [dl, dl, ..., dl]  $\rightarrow$  Registro máscara el equipo  
 pxor xmm2, xmm2  $\rightarrow$  Registro máscara acierto  
 pxor xmm3, xmm3  $\rightarrow$  Registro máscara equipo  
 pxor xmm4, xmm4  $\rightarrow$  Registro acumulador lanz

ciclo:  
 movdqu xmm0, [rdi]  
 movdqu xmm2, xmm0  
 movdqu xmm3, xmm0  
 psllw xmm0, 14  $\rightarrow$  xmm0 = [lanz10, lanz10, ..., lanz10]  
 psrlw xmm0, 14  $\rightarrow$  xmm0 = [lanz, lanz, ..., lanz]  $\rightarrow$  Registro lanzamientos  
 psllw xmm2, 13  $\rightarrow$  xmm2 = [acertolanz10, ..., acertolanz10]  
 psraw xmm2, 15  $\rightarrow$  xmm2 = [acierto? FFFF:0000, ..., acierto? FFFF:0000]  
 psllw xmm3, 12  $\rightarrow$  xmm3 = [equipo/acertolanz10, ..., equipo/acertolanz10]  
 psrlw xmm3, 15  $\rightarrow$  xmm3 = [equipo, equipo, ..., equipo]  
 psrqrw xmm3, xmm1  $\rightarrow$  xmm3 = [equipo == el equipo? FFFF:0000, ..., equipo == el equipo? FFFF:0000]  
 pand xmm3, xmm2  $\rightarrow$  xmm3 = [equipo == el equipo && acertolanz? FFFF:0000, ..., equipo == el equipo && acertolanz? FFFF:0000]  
 pand xmm3, xmm0  $\rightarrow$  xmm3 = [lanz, 0000, ..., lanz, 0000]  
 paddw xmm4, xmm3  $\rightarrow$  Acumula lanz en xmm4  
 lea rdi, [rdi+8]  $\rightarrow$  Cerra el puntero a lanzamientos  
 loop ciclo  $\rightarrow$  Decrece ecx y hace ciclo o no  
 // xmm4 = [lanz15, lanz14, ..., lanz0]  
 phaddw xmm4, xmm4  $\rightarrow$  xmm4 = [lanz15+lanz14, lanz13+lanz12, ..., lanz1+lanz0]  
 phaddw xmm4, xmm4  $\rightarrow$  xmm4 = [..., lanz15+lanz14+lanz13+lanz12, ..., lanz3+lanz2+lanz1+lanz0]  
 phaddw xmm4, xmm4  $\rightarrow$  xmm4 = [..., lanz15+lanz14+...+lanz3, lanz2+lanz1+...+lanz0]  
 phaddw xmm4, xmm4  $\rightarrow$  xmm4 = [..., lanz15+lanz14+...+lanz1+lanz0]  
 movd eax, xmm4  $\rightarrow$  eax = lanz15+lanz14+...+lanz1+lanz0

b) Como el promedio entre todos los lanzamientos (acertados o no) y los triples (acertados o no), de la forma:  $\frac{\text{triples}}{\text{totales}}$ , siendo estos de local:

triples\_mask dw 3h, 3h, 3h, 3h, 3h, 3h, 3h, 3h  $\rightarrow$  Máscara de triples

promedio de tres:  
 push rbp  
 mov rbp, rsp  
 pxor xmm0, xmm0  $\rightarrow$  Registro lanzamientos  
 pxor xmm1, xmm1  $\rightarrow$  Registro locales  
 movdqu xmm2, [triples\_mask]  $\rightarrow$  Registro máscara triples  
 pxor xmm3, xmm3  $\rightarrow$  Registro acumulador locales



pxor xmm4, xmm4

→ Registro acumulador Triples locales

mov ecx, esi

→ ECX: Contador del ciclo

shr ecx, 3

→ El contador avanza de 8 posiciones (tamaño múltiplo de 8)

ciclo:

mordq xmm0, [rdi]

→ xmm0 = [jug1.equipo/acierto/lanz, ..., jug1.equipo/acierto/lanz]

mordq xmm1, xmm0

psllw xmm0, 14

→ xmm0 = [lanz/0, ..., lanz/0]

psrlw xmm0, 14

→ xmm0 = [lanz, ..., lanz]

pcmpgq xmm0, xmm2

→ xmm0 = [lanz == 3h? FFFF:0000, ..., lanz == 3h? FFFF:0000]

psllw xmm1, 12

→ xmm1 = [equipo/acierto/lanz, ..., equipo/acierto/lanz]

psrlw xmm1, 15

→ xmm1 = [equipo, ..., equipo]

pand xmm0, xmm1

→ xmm0 = [lanz == 3h? equipo:0000, ..., lanz == 3h? equipo:0000]

paddw xmm3, xmm1

→ Acumulo en xmm3 los lanzamientos locales (equipo: 1 ⇔ equipo: local)

paddw xmm4, xmm0

→ Acumulo en xmm4 los lanzamientos triples locales

lea rdi, [rdi+16]

→ Avanzo rdi a los siguientes 8 lanzamientos

loop .ciclo

→ Decremento ecx e itero ciclo o no según si llega a 0

// xmm3 = [locales<sub>15</sub>, locales<sub>14</sub>, ..., locales<sub>1</sub>, locales<sub>0</sub>]; xmm4 = [locales\_triples<sub>15</sub>, ..., locales\_triples<sub>0</sub>]

phaddw xmm3, xmm3

phaddw xmm3, xmm3

phaddw xmm3, xmm3

phaddw xmm3, xmm3

→ xmm3 = [..., locales<sub>15</sub> + locales<sub>14</sub> + ... + locales<sub>1</sub> + locales<sub>0</sub>] (análogo al proceso de xmm4 en 2a))

phaddw xmm4, xmm4

phaddw xmm4, xmm4

phaddw xmm4, xmm4

phaddw xmm4, xmm4

→ xmm4 = [..., locales\_triples<sub>15</sub> + locales\_triples<sub>14</sub> + ... + locales\_triples<sub>0</sub>]

pslld xmm3, 16

psrlq xmm3, 16

pslld xmm4, 16

psrlq xmm4, 16

→ xmm3 = [..., 0, locales<sub>15</sub> + locales<sub>14</sub> + ... + locales<sub>0</sub>] (extensión de 16b a 32b)

→ xmm4 = [..., 0, locales\_triples<sub>15</sub> + locales\_triples<sub>14</sub> + ... + locales\_triples<sub>0</sub>] (extensión de 16b a 32b)

cvtss2ss xmm3, xmm3

→ xmm3 = [..., float(locales<sub>15</sub> + ... + locales<sub>0</sub>)]

cvtss2ss xmm4, xmm4

→ xmm4 = [..., float(locales\_triples<sub>15</sub> + ... + locales\_triples<sub>0</sub>)]

divss xmm4, xmm3

→ xmm4 = [..., float(locales\_triples<sub>15</sub> + ... + locales\_triples<sub>0</sub>) / float(locales<sub>15</sub> + ... + locales<sub>0</sub>)]

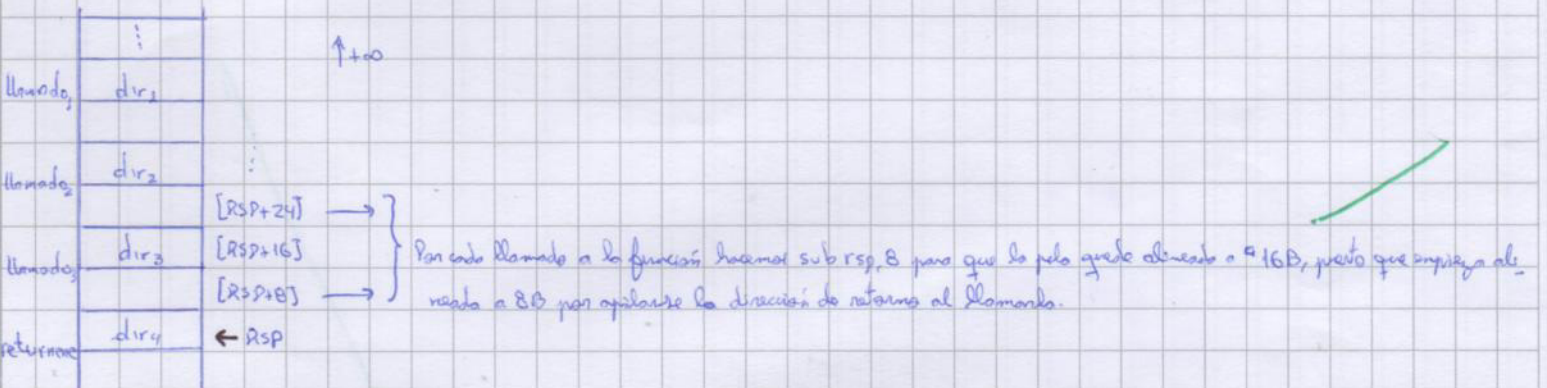
pxor xmm0, xmm0

movss xmm0, xmm4

ret

→ xmm0 = [..., promedio entre los triples locales y los locales finales]

3 a) Siendo que la función `returnene` no utiliza variables locales ni preserva registros, por cada llamado a la función solo se apila la dirección de retorno del punto en que se llama. Suponiendo que al hacer el llamado cada vez a la función la pila debe estar alineada a 16B y que la función se llama por primera vez desde `dir1`, luego `dir2` y luego `dir3`, resta que la pila antes de llamar a `returnene` (desde `dir4`) es como la que sigue:



b) `void returnene (unsigned int n); → edi = n`

`returnene:`

```

mov r8, rsp      → Puntero a la pila que empieza en rsp que es la dir de retorno de returnene
mov ecx, edi     → Contador de n = 0

ciclo:
add r8, 16       → r8 = r8 + 16 (r8 apunta dos posiciones más arriba en la pila, pasando a la dir exterior a la que apuntaba)
loop ciclo

sub rsp, 8       → Alinea la pila para hacer el llamado
call [r8]        → Llamo a la función con su dirección

add rsp, 8       → Restaura la pila para retornar (si eventualmente lo hace)
ret
  
```

En que RSP está ahí solo hace RET

La función sigue al superior de la pila de 3a) y partiendo de la dirección del punto en que se llamó `returnene` y recorriendo la pila hasta hallar el punto en que está la dirección de retorno de la función anterior.